Automatic Parallelization of Divide and Conquer Algorithms

Radu Rugina and Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology

Outline

- Example
- Information required to parallelize divide and conquer algorithms
- How compiler extracts parallelism
  - Key technique: constraint systems
- Results
- Related work
- Conclusion

Example - Divide and Conquer Sort

7	4	6	1	3	5	8	2

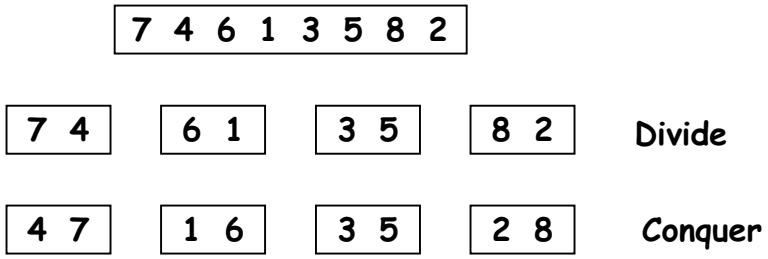
Example - Divide and Conquer Sort

7	4	6	1	3	5	8	2

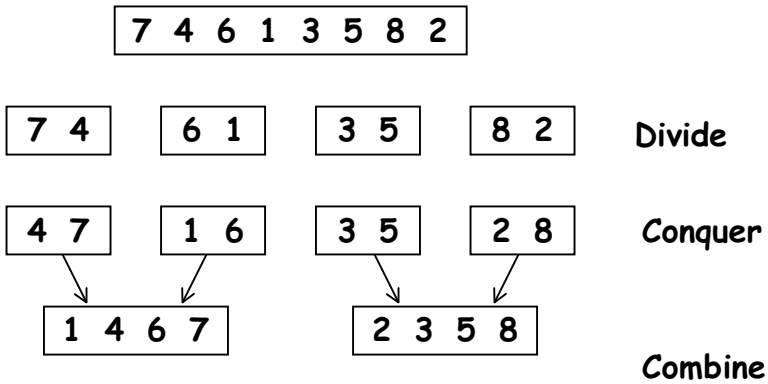
7	4		6	1		3	5		8	2	

Divide

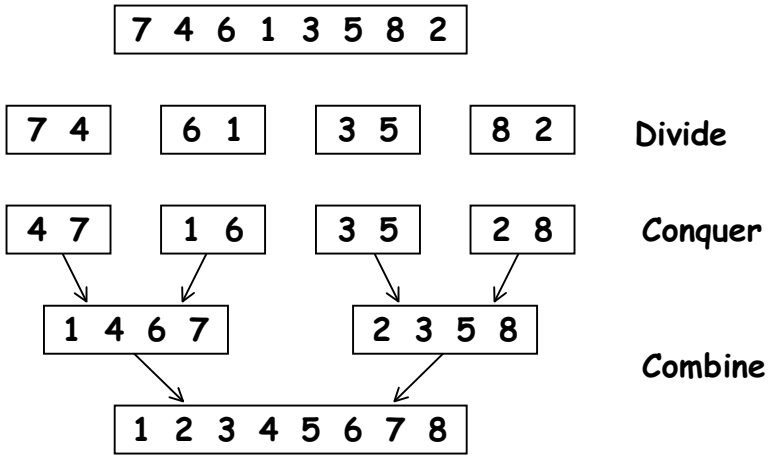
Example - Divide and Conquer Sort



Example - Divide and Conquer Sort



Example - Divide and Conquer Sort

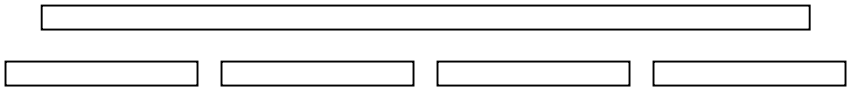


Divide and Conquer Algorithms

- Lots of Generated Concurrency
  - Solve Subproblems in Parallel

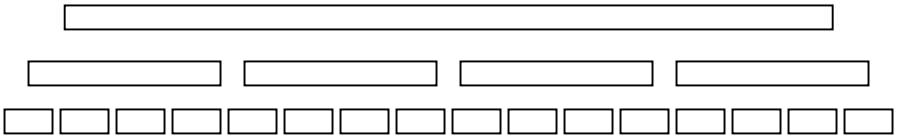
Divide and Conquer Algorithms

- Lots of Generated Concurrency
  - Solve Subproblems in Parallel



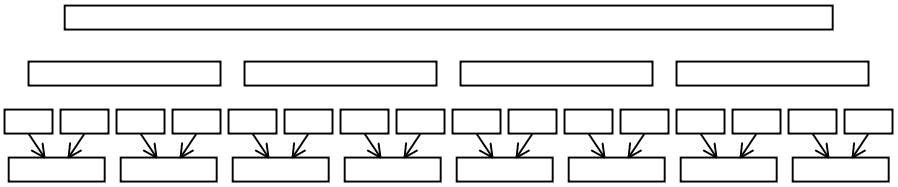
Divide and Conquer Algorithms

- Lots of Recursively Generated Concurrency
  - Recursively Solve Subproblems in Parallel



Divide and Conquer Algorithms

- Lots of Recursively Generated Concurrency
  - Recursively Solve Subproblems in Parallel
  - Combine Results in Parallel



Divide and Conquer Algorithms

- Lots of Recursively Generated Concurrency
  - Recursively Solve Subproblems in Parallel
  - Combine Results in Parallel
- Good Cache Performance
  - Problems Naturally Scale to Fit in Cache
  - No Cache Size Constants in Code

Divide and Conquer Algorithms

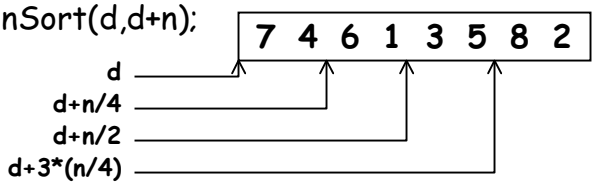
- Lots of Recursively Generated Concurrency
  - Recursively Solve Subproblems in Parallel
  - Combine Results in Parallel
- Good Cache Performance
  - Problems Naturally Scale to Fit in Cache
  - No Cache Size Constants in Code
- Lots of Programs
  - Sort Programs
  - Dense Matrix Programs

```
  "Sort n Items in d, Using t as Temporary Storage"
void sort(int *d, int *t, int n)
    if (n > CUTOFF) {
       sort(d,t,n/4);
       sort(d+n/4,t+n/4,n/4);
       sort(d+n/2,t+n/2,n/4);
       sort(d+3*(n/4),t+3*(n/4),n-3*(n/4));
       merge(d,d+n/4,d+n/2,t);
       merge(d+n/2,d+3*(n/4),d+n,t+n/2);
       merge(t,t+n/2,t+n,d);
   } else insertionSort(d,d+n);
```

"Recursively Sort Four Quarters of d"

```
void sort(int *d, int *t, int n)
    if (n > CUTOFF) {
  ⇨ sort(d,t,n/4);
  ⇨ sort(d+n/4,t+n/4,n/4);
  ⇨ sort(d+n/2,t+n/2,n/4);
  ⇨ sort(d+3*(n/4),t+3*(n/4),n-3*(n/4));
    merge(d,d+n/4,d+n/2,t);
    merge(d+n/2,d+3*(n/4),d+n,t+n/2);
    merge(t,t+n/2,t+n,d);
  } else insertionSort(d,d+n);
```

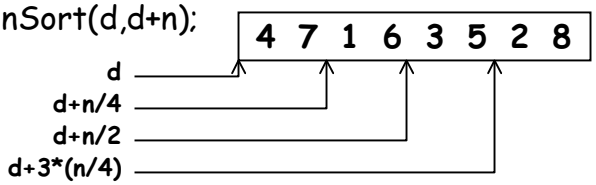
Subproblems Identified Using Pointers Into Middle of Array



"Recursively Sort Four Quarters of d"

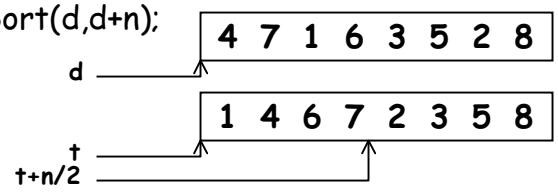
```
void sort(int *d, int *t, int n)
    if (n > CUTOFF) {
  ⇨ sort(d,t,n/4);
  ⇨ sort(d+n/4,t+n/4,n/4);
  ⇨ sort(d+n/2,t+n/2,n/4);
  ⇨ sort(d+3*(n/4),t+3*(n/4),n-3*(n/4));
    merge(d,d+n/4,d+n/2,t);
    merge(d+n/2,d+3*(n/4),d+n,t+n/2);
    merge(t,t+n/2,t+n,d);
  } else insertionSort(d,d+n);
```

Sorted Results Written Back Into Input Array



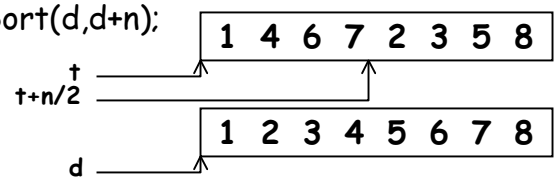
"Merge Sorted Quarters of d Into Halves of t"

```
void sort(int *d, int *t, int n)
    if (n > CUTOFF) {
       sort(d,t,n/4);
       sort(d+n/4,t+n/4,n/4);
       sort(d+n/2,t+n/2,n/4);
       sort(d+3*(n/4),t+3*(n/4),n-3*(n/4));
    ⇨ merge(d,d+n/4,d+n/2,t);
    ⇨ merge(d+n/2,d+3*(n/4),d+n,t+n/2);
       merge(t,t+n/2,t+n,d);
    } else insertionSort(d,d+n);
```



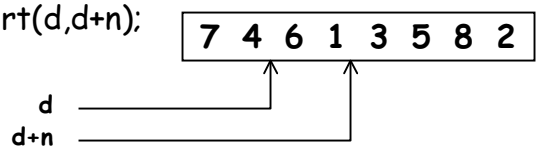
"Merge Sorted Halves of t Back Into d"

```
void sort(int *d, int *t, int n)
    if (n > CUTOFF) {
       sort(d,t,n/4);
       sort(d+n/4,t+n/4,n/4);
       sort(d+n/2,t+n/2,n/4);
       sort(d+3*(n/4),t+3*(n/4),n-3*(n/4));
       merge(d,d+n/4,d+n/2,t);
       merge(d+n/2,d+3*(n/4),d+n,t+n/2);
    ⇨ merge(t,t+n/2,t+n,d);
    } else insertionSort(d,d+n);
```



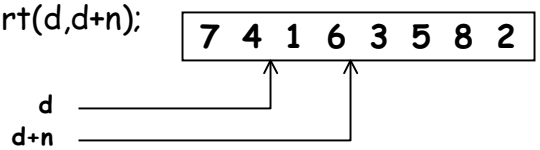
"Use a Simple Sort for Small Problem Sizes"

```
void sort(int *d, int *t, int n)
    if (n > CUTOFF) {
       sort(d,t,n/4);
       sort(d+n/4,t+n/4,n/4);
       sort(d+n/2,t+n/2,n/4);
       sort(d+3*(n/4),t+3*(n/4),n-3*(n/4));
       merge(d,d+n/4,d+n/2,t);
       merge(d+n/2,d+3*(n/4),d+n,t+n/2);
       merge(t,t+n/2,t+n,d);
⇨  } else insertionSort(d,d+n);
```



"Use a Simple Sort for Small Problem Sizes"

```
void sort(int *d, int *t, int n)
    if (n > CUTOFF) {
       sort(d,t,n/4);
       sort(d+n/4,t+n/4,n/4);
       sort(d+n/2,t+n/2,n/4);
       sort(d+3*(n/4),t+3*(n/4),n-3*(n/4));
       merge(d,d+n/4,d+n/2,t);
       merge(d+n/2,d+3*(n/4),d+n,t+n/2);
       merge(t,t+n/2,t+n,d);
⇨  } else insertionSort(d,d+n);
```



Parallel Execution

```
void sort(int *d, int *t, int n)
    if (n > CUTOFF) {
       spawn sort(d,t,n/4);
       spawn sort(d+n/4,t+n/4,n/4);
       spawn sort(d+n/2,t+n/2,n/4);
       spawn sort(d+3*(n/4),t+3*(n/4),n-3*(n/4));
       sync;
       spawn merge(d,d+n/4,d+n/2,t);
       spawn merge(d+n/2,d+3*(n/4),d+n,t+n/2);
       sync;
       merge(t,t+n/2,t+n,d);
   } else insertionSort(d,d+n);
```

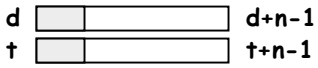
What Do You Need to Know to Exploit this Form of Parallelism?

What Do You Need to Know to Exploit this Parallelism?

Calls to sort access disjoint parts of d and t

Together, calls access [d,d+n-1] and [t,t+n-1]

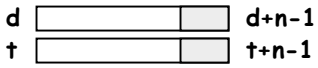
sort(d,t,n/4);



sort(d+n/4,t+n/4,n/4);

sort(d+n/2,t+n/2,n/4);

sort(d+3*(n/4),t+3*(n/4),n-3*(n/4));

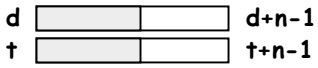


What Do You Need to Know to Exploit this Parallelism?

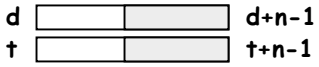
First two calls to merge access disjoint parts of d,t

Together, calls access [d,d+n-1] and [t,t+n-1]

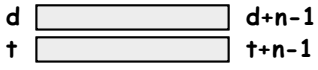
merge(d,d+n/4,d+n/2,t);



merge(d+n/2,d+3*(n/4),d+n,t+n/2);



merge(t,t+n/2,t+n,d);



What Do You Need to Know to Exploit this Parallelism?

Calls to insertionSort access [d,d+n-1]

insertionSort(d,d+n);

What Do You Need to Know to Exploit this Parallelism?

The Regions of Memory Accessed by Complete Executions of Procedures

How Hard Is it to Extract these Regions?

How Hard Is it to Extract these Regions?

Challenging

How Hard Is it to Extract these Regions?

```
insertionSort(int *l, int *h) {
  int *p, *q, k;
  for (p = l+1; p < h; p++) {
      for (k = *p, q = p-1; l <= q && k < *q; q--)
            *(q+1) = *q;
      *(q+1) = k;
  }
}
```

Not Immediately Obvious That insertionSort(l,h) Accesses [l,h-1]

How Hard Is it to Extract these Regions?

```
merge(int *l1, int*m, int *h2, int *d) {
  int *h1 = m; int *l2 = m;
  while ((l1 < h1) && (l2 < h2))
      if (*l1 < *l2) *d++ = *l1++;
      else *d++ = *l2++;
  while (l1 < h1) *d++ = *l1++;
  while (l2 < h2) *d++ = *l2++;
}
```

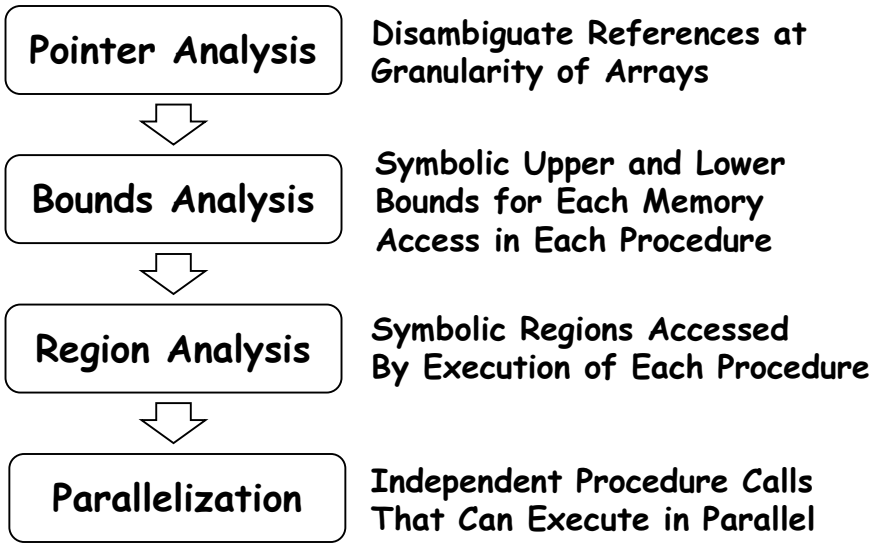
Not Immediately Obvious That merge(l,m,h,d) Accesses [l,h-1] and [d,d+(h-l)-1]

Issues

- Pervasive Use of Pointers
  - Pointers into Middle of Arrays
  - Pointer Arithmetic
  - Pointer Comparison
- Multiple Procedures
  - sort(int *d, int *t, n)
  - insertionSort(int *l, int *h)
  - merge(int *l, int *m, int *h, int *t)
- Recursion

How The Compiler Does It

Structure of Compiler



Example

```
f(char *p, int n)
  if (n > CUTOFF) {
      f(p, n/2);          initialize first half
      f(p+n/2, n/2);      initialize second half
  } else {
      base case: initialize small array
      int i = 0;
      while (i < n) { *(p+i) = 0; i++; }
  }
```

Bounds Analysis

- For each variable at each program point, derive upper and lower bounds for value
- Bounds are symbolic expressions
  - symbolic variables in expressions represent initial values of parameters
  - linear combinations of these variables
  - multivariate polynomials

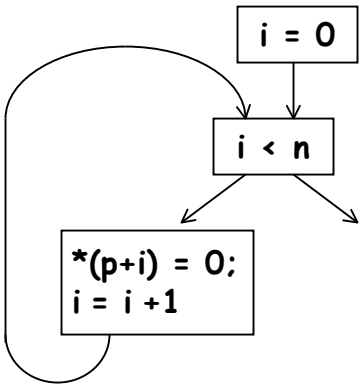
Bounds Analysis

What are upper and lower bounds for region accessed by while loop in base case?

```
int i = 0;
while (i < n) { *(p+i) = 0; i++; }
```

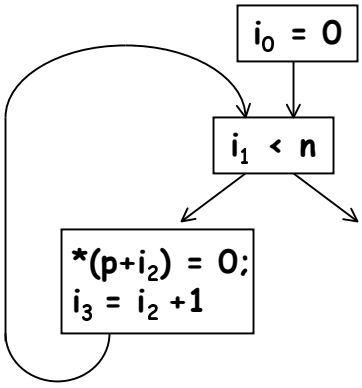
Bounds Analysis, Step 1

Build control flow graph



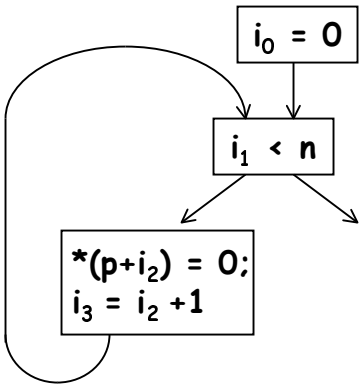
Bounds Analysis, Step 2

Number different versions of variables



Bounds Analysis, Step 3

Set up constraints for lower bounds



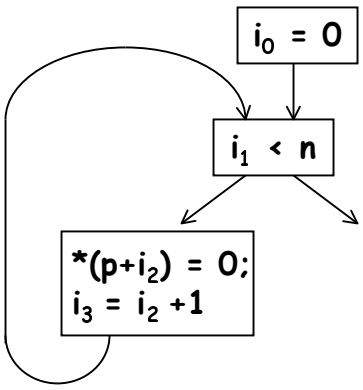
$l(i_0) <= 0$

$l(i_1) <= l(i_0)$
$l(i_1) <= l(i_3)$

$l(i_2) <= l(i_1)$
$l(i_3) <= l(i_2)+1$

Bounds Analysis, Step 3

Set up constraints for lower bounds



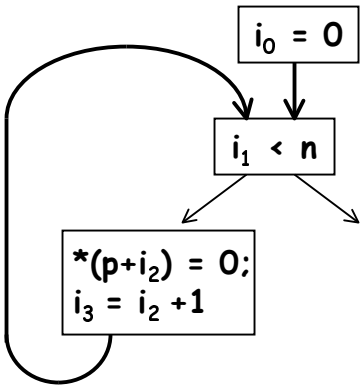
$l(i_0) <= 0$ ⇦

$l(i_1) <= l(i_0)$
$l(i_1) <= l(i_3)$

$l(i_2) <= l(i_1)$
$l(i_3) <= l(i_2)+1$

Bounds Analysis, Step 3

Set up constraints for lower bounds



$l(i_0) <= 0$

$l(i_1) <= l(i_0)$ ⇦

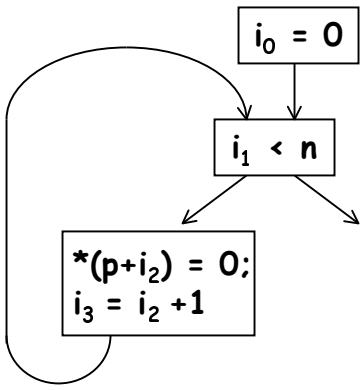
$l(i_1) <= l(i_3)$ ⇦

$l(i_2) <= l(i_1)$

$l(i_3) <= l(i_2)+1$

Bounds Analysis, Step 4

Set up constraints for upper bounds



$l(i_0) <= 0$

$l(i_1) <= l(i_0)$

$l(i_1) <= l(i_3)$

$l(i_2) <= l(i_1)$

$l(i_3) <= l(i_2)+1$

$0 <= u(i_0)$

$u(i_0) <= u(i_1)$

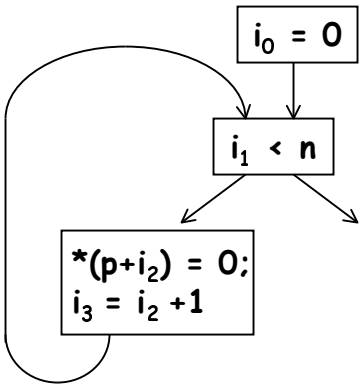
$u(i_3) <= u(i_1)$

$\min(u(i_1),n-1) <= u(i_2)$

$u(i_2)+1 <= u(i_3)$

Bounds Analysis, Step 4

Set up constraints for upper bounds



$l(i_0) <= 0$

$l(i_1) <= l(i_0)$
$l(i_1) <= l(i_3)$

$l(i_2) <= l(i_1)$
$l(i_3) <= l(i_2)+1$

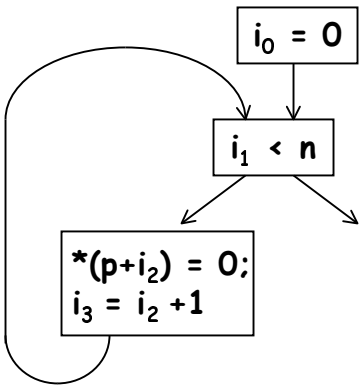
$0 <= u(i_0)$

$u(i_0) <= u(i_1)$
$u(i_3) <= u(i_1)$

$\min(u(i_1),n-1) <= u(i_2)$ ⇦
$u(i_2)+1 <= u(i_3)$

Bounds Analysis, Step 4

Set up constraints for upper bounds



$l(i_0) <= 0$

$l(i_1) <= l(i_0)$
$l(i_1) <= l(i_3)$

$l(i_2) <= l(i_1)$
$l(i_3) <= l(i_2)+1$

$0 <= u(i_0)$

$u(i_0) <= u(i_1)$
$u(i_3) <= u(i_1)$

$n-1 <= u(i_2)$ ⇦
$u(i_2)+1 <= u(i_3)$

Bounds Analysis, Step 5

Generate symbolic expressions for bounds

Goal: express bounds in terms of parameters

$$l(i_0) = c_1p + c_2n + c_3$$
$$l(i_1) = c_4p + c_5n + c_6$$
$$l(i_2) = c_7p + c_8n + c_9$$
$$l(i_3) = c_{10}p + c_{11}n + c_{12}$$

$$u(i_0) = c_{13}p + c_{14}n + c_{15}$$
$$u(i_1) = c_{16}p + c_{17}n + c_{18}$$
$$u(i_2) = c_{19}p + c_{20}n + c_{21}$$
$$u(i_3) = c_{22}p + c_{23}n + c_{24}$$

Bounds Analysis, Step 6

Substitute expressions into constraints

$$c_1p + c_2n + c_3 \le 0$$

$$c_4p + c_5n + c_6 \le c_1p + c_2n + c_3$$
$$c_4p + c_5n + c_6 \le c_{10}p + c_{11}n + c_{12}$$

$$c_7p + c_8n + c_9 \le c_4p + c_5n + c_6$$
$$c_{10}p + c_{11}n + c_{12} \le c_7p + c_8n + c_9 + 1$$

$$0 \le c_{13}p + c_{14}n + c_{15}$$

$$c_{13}p + c_{14}n + c_{15} \le c_{16}p + c_{17}n + c_{18}$$
$$c_{22}p + c_{23}n + c_{24} \le c_{16}p + c_{17}n + c_{18}$$

$$n - 1 \le c_{19}p + c_{20}n + c_{21}$$
$$c_{19}p + c_{20}n + c_{21} + 1 \le c_{22}p + c_{23}n + c_{24}$$

Goal

Solve Symbolic Constraint System

find values for constraint variables c_1, ..., c_{24}
that satisfy the inequality constraints

Maximize Lower Bounds

Minimize Upper Bounds

Bounds Analysis, Step 7

Apply expression ordering principle

$c_1p + c_2n + c_3 <= c_4p + c_5n + c_6$

If

$c_1 <= c_4$, $c_2 <= c_5$, and $c_3 <= c_6$

Bounds Analysis, Step 7

Apply expression ordering principle

Generate a linear program

$c_1 <= 0$ $c_2 <= 0$ $c_3 <= 0$ $0 <= c_{13}$ $0 <= c_{14}$ $0 <= c_{15}$

$c_4 <= c_1$ $c_5 <= c_2$ $c_6 <= c_3$ $c_{13} <= c_{16}$ $c_{14} <= c_{17}$ $c_{15} <= c_{18}$

$c_4 <= c_{10}$ $c_5 <= c_{11}$ $c_6 <= c_{12}$ $c_{22} <= c_{16}$ $c_{23} <= c_{17}$ $c_{24} <= c_{18}$

$c_7 <= c_4$ $c_8 <= c_5$ $c_9 <= c_6$ $0 <= c_{19}$ $1 <= c_{20}$ $-1 <= c_{21}$

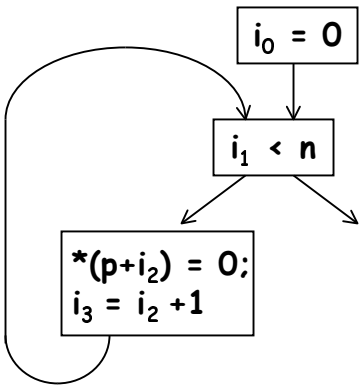
$c_{10} <= c_7$ $c_{11} <= c_8$ $c_{12} <= c_9+1$ $c_{19} <= c_{22}$ $c_{20} <= c_{23}$ $c_{21}+1 <= c_{24}$

Objective Function:

$$\max \underbrace{(c1 + \cdots + c12)}_{\text{lower bounds}} - \underbrace{(c13 + \cdots + c24)}_{\text{upper bounds}}$$

Bounds Analysis, Step 8

Solve linear program to extract bounds



$l(i_0) = 0$
$l(i_1) = 0$
$l(i_2) = 0$
$l(i_3) = 0$

$u(i_0) = 0$
$u(i_1) = n$
$u(i_2) = n-1$
$u(i_3) = n$

Region Analysis

Goal: Compute Accessed Regions of Memory

- Intra-Procedural
  - Use bounds at each load or store
  - Compute accessed region
- Inter-Procedural
  - Use intra-procedural results
  - Set up another constraint system
  - Solve to find regions accessed by entire execution of the procedure

Basic Principle of Inter-Procedural Region Analysis

- For each procedure
  - Generate symbolic expressions for upper and lower bounds of accessed regions
- Constraint System
  - Accessed regions include regions accessed by statements in procedure
  - Accessed regions include regions accessed by invoked procedures

Inter-Procedural Constraints in Example

```
f(char *p, int n)
  if (n > CUTOFF) {
      f(p, n/2);                    l(f,p,n) <= l(f,p,n/2)
                                    u(f,p,n) <= u(f,p,n/2)
      f(p+n/2, n/2);                l(f,p,n) <= l(f,p+n/2,n/2)
                                    u(f,p,n) <= u(f,p+n/2,n/2)
  } else {
      int i = 0;
      while (i < n) {
         *(p+i) = 0; i++;           l(f,p,n) <= p
      }                             u(f,p,n) <= p+n-1
  }
```

Derive Constraint System

- Generate symbolic expressions
  - $l(f,p,n) = C_1p + C_2n + C_3$
  - $u(f,p,n) = C_4p + C_5n + C_6$
- Build constraint system
  - $C_1p + C_2n + C_3 <= p$
  - $C_4p + C_5n + C_6 <= p + n - 1$
  - $C_1p + C_2n + C_3 <= C_1p + C_2(n/2) + C_3$
  - $C_4p + C_5n + C_6 <= C_4p + C_5(n/2) + C_6$
  - $C_1p + C_2n + C_3 <= C_1(p+n/2) + C_2(n/2) + C_3$
  - $C_4p + C_5n + C_6 <= C_4(p+n/2) + C_5(n/2) + C_6$

Solve Constraint System

- Simplify Constraint System
  - $C_1p + C_2n + C_3 <= p$
  - $C_4p + C_5n + C_6 <= p + n - 1$
  - $C_2n <= C_2(n/2)$
  - $C_5n <= C_5(n/2)$
  - $C_2(n/2) <= C_1(n/2)$
  - $C_5(n/2) <= C_4(n/2)$
- Generate and Solve Linear Program
  - l(f,p,n) = p
  - u(f,p,n) = p+n-1

Parallelization

- Dependence Testing of Two Calls
  - Do accessed regions intersect?
  - Based on comparing upper and lower bounds of accessed regions
  - Comparison done using expression ordering principle
- Parallelization
  - Find sequences of independent calls
  - Execute independent calls in parallel

Details

- Inter-procedural positivity analysis
  - Verify that variables are positive
  - Required for correctness of expression ordering principle
- Correlation Analysis
- Integer Division
  - Basic Idea : $(n-1)/2 <= \lfloor n/2 \rfloor <= n/2$
  - Generalized : $(n-m+1)/m <= \lfloor n/m \rfloor <= n/m$
- Linear System Decomposition

Experimental Results

- Implementation - SUIF, lp_solve, Cilk

Speedup for Matrix Multiply

Speedup for Sort

Thanks: Darko Marinov, Nate Kushman, Don Dailey

Related Work

- Shape Analysis
  - Chase, Wegman, Zadek (PLDI 90)
  - Ghiya, Hendren (POPL 96)
  - Sagiv, Reps, Wilhelm (TOPLAS 98)
- Commutativity Analysis
  - Rinard and Diniz (PLDI 96)
- Predicated Dataflow Analysis
  - Moon, Hall, Murphy (ICS 98)

Related Work

- Array Region Analysis
  - Triolet, Irigoin and Feautrier (PLDI 86)
  - Havlak and Kennedy (IEEE TPDS 91)
  - Hall, Amarasinghe, Murphy, Liao and Lam (SC 95)
  - Gu, Li and Lee (PPoPP 97)
- Symbolic Analysis of Loop Variables
  - Blume and Eigenmann (IPPS 95)
  - Haghigat and Polychronopoulos (LCPC 93)

Future

- Static Race Detection for Explicitly Parallel Programs
- Static Elimination of Array Bounds Checks
- Static Pointer Validation Checks
- Result:
  - Safety Guarantees
  - No Efficiency Compromises

Context

- Mainstream Parallelizing Compilers
  - Loop Nests, Dense Matrices
  - Affine Access Functions
  - Key Problem:Solving Diophantine Equations
- Compilers for Divide and Conquer Algorithms
  - Recursion, Dense Arrays (dynamic)
  - Pointers, Pointer Arithmetic
  - Key Problems: Pointer Analysis, Symbolic Region Analysis, Solving Linear Programs